Génération automatique de simulateurs fonctionnels de processeurs

Tahiry Ratsiambahotra, Hugues Cassé, Christine Rochange, Pascal Sainrat

Génération automatique de simulateurs fonctionnels de processeurs

Tahiry Ratsiambahotra, Hugues Cassé, Christine Rochange, Pascal Sainrat,

Institut de Recherche en Informatique de Toulouse
Université de Toulouse - CNRS
31062 Toulouse cedex 9, France
http://www.irit.fr/TRACES
{ratsiam, casse, rochange, sainrat }@irit.fr

Résumé
Le développement d'un simulateur de processeur est long et fastidieux. Découpler la partie fonctionnelle (émulation) de la partie structure (analyse des temps de traitement) permet de réutiliser plus facilement du code existant (principalement le code d'émulation, les jeux d'instructions évoluant moins vite que les architectures matérielles). Dans ce contexte, plusieurs équipes ont proposé des solutions pour une génération automatique de la partie fonctionnelle d'un simulateur à partir d'une description plus ou moins formelle du jeu d'instructions. S'il est relativement aisé de générer automatiquement un émulateur pour l'architecture DLX, il s'avère plus compliqué de réaliser un générateur supportant à la fois des architectures de type CISC, RISC ou VLIW et produisant un code efficace. Dans cet article, nous décrivons plusieurs techniques mises en œuvre dans l'outil GLISS que nous avons développé et qui se veut aussi « polyvalent » que possible.

Mots-clés : jeux d'instructions, simulation, test

1. Introduction

De plus en plus, les systèmes embarqués font usage de plusieurs processeurs de caractéristiques différentes qui évoluent rapidement selon les besoins en performance croissants du marché. En même temps, la vitesse de mise sur le marché d'un nouveau produit devient un facteur déterminant de compétitivité. Ceci nécessite de disposer d'outils de simulation des architectures retenues afin de pouvoir tester et améliorer suffisamment tôt le logiciel, avant même que le matériel n'ait été complètement développé. La simulation peut également permettre de comparer plusieurs architectures et contribuer ainsi à l'efficacité du processus de conception.

De nombreux simulateurs de processeurs ont été et sont développés, que ce soit par les constructeurs eux-mêmes ou par des équipes de recherche académiques. Parmi ces derniers, on peut citer SimpleScalar [2], UniSim [1] ou encore le simulateur que nous avons développé dans le cadre de la plateforme OTAWA [3] dédiée au calcul de temps d'exécution pire-cas. Quelle que soit l'architecture cible, le simulateur comporte généralement deux composantes qui, selon l'approche du développeur, peuvent être imbriquées ou, au contraire, complètement découplées comme c'est de plus en plus souvent le cas. La première de ces composantes, que nous appellerons *simulateur fonctionnel*, prend en charge la lecture des codes opérations, leur décodage et leur émulation. La seconde, désignée par *simulateur structurel*, prend en compte les caractéristiques du matériel cible pour déterminer le temps d'exécution des instructions et du programme. Bien que ces deux composantes présentent les mêmes fonctionnalités d'un simulateur à l'autre, elles sont en général complètement réécrites lorsque l'on produit un nouveau simulateur. De ce fait, le développement et la mise au point d'un simulateur de processeur sont généralement laborieux et très coûteux en temps. C'est pourquoi un certain

nombre de travaux de recherche visent à définir les outils nécessaires à la génération automatique de simulateurs fonctionnels et structurels à partir d'une description plus ou moins formelle de l'architecture. L'objectif est de restreindre la tâche du concepteur à la seule description des caractéristiques propres de l'architecture tandis que le code implémentant les mécanismes généraux de la simulation est généré automatiquement.

Dans cet article, nous nous intéressons à la génération automatique de simulateurs fonctionnels (simulateurs de jeu d'instructions, *instruction set simulators* ou *ISS*) et nous présentons notre outil GLISS (*Generating Libraries for Instruction Set Simulation*). Cet outil génère une bibliothèque de fonctions clés pour la simulation fonctionnelle à partir de la description d'un jeu d'instructions en langage GLISS-nML. Ce langage a été construit à partir du langage de description d'architectures nML [5] et permet la modélisation d'une plus grande variété de jeux d'instructions grâce à l'introduction de concepts améliorant la flexibilité.

Dans la section 2, nous présentons le langage nML qui est à l'origine de nos travaux. Nous expliquons, dans la section 3, comment il est possible d'exploiter une description nML pour générer un simulateur fonctionnel. Dans la section 4, nous introduirons le langage GLISS-nML qui comble les lacunes de nML en termes de flexibilité. Nous présenterons GLISS, notre générateur automatique de simulateurs fonctionnels, dans la section 5. Nous conclurons dans la section 6.

2. Génération d'un simulateur fonctionnel à partir de la description d'un jeu d'instructions en langage nML

2.1 Le langage nML

Le langage nML a été proposé dès 1991 pour permettre la description formelle de jeux d'instructions [5]. Il se présente sous la forme d'une grammaire attribuée qui permet une description hiérarchique des instructions. Nous allons présenter ici les principaux éléments du langage.

Un jeu d'instructions (ou ISA : *Instruction Set Architecture*) est composé de trois éléments : la structure des registres et de l'espace d'adressage en mémoire, les modes d'adressage qui permettent de désigner des opérandes, et les instructions proprement dites. Ces trois éléments se retrouvent dans une description nML.

2.1.1 Déclarations

La première partie de la description d'un jeu d'instructions est constituée de déclarations qui définissent les composantes de l'état interne du processeur. Il s'agit de spécifier la liste des registres et leurs caractéristiques, ainsi que la mémoire. Ceci est illustré par les lignes 7-9 du code de la Figure 1. La partie « déclarations » peut également contenir des définitions de types (lignes 3-5) ou des déclarations de constantes (lignes 1-2) ou de variables temporaires (ligne 10).

2.1.2 Spécifications des modes d'adressage

Ils sont utilisés pour désigner les opérandes. Chaque mode d'adressage a deux attributs qui expriment comment il est représenté en assembleur (attribut `syntax`) et en binaire (attribut `image`).

La Figure 1 (lignes 12-17) donne les définitions de deux modes d'adressage. Le mot-clé `mode` indique que l'on va décrire un mode d'adressage ; il est suivi d'un identifiant du mode et d'une liste de paramètres. Chacun des deux modes est décrit comme une règle ET. La manière dont la valeur de l'opérande est obtenue à partir de l'état du processeur et de la valeur des paramètres est définie après

le signe « égal ». Les attributs du mode d'adressage peuvent être spécifiés à l'aide de la fonction `format` dont le premier argument a une syntaxe dérivée des formats du langage C. Par exemple, la ligne 13 indique que le mode d'adressage REGISTER est désigné dans le langage d'assemblage par une chaîne de caractères du type « `r%d` » où `%d` représente le numéro du registre. La ligne 14 signifie que ce mode d'adressage est représenté, dans le code binaire d'une instruction, par 7 bits, dont les deux premiers ont la valeur `01` et les cinq suivants codent le numéro de registre. Le mode ADDR (ligne 18) est lui défini par une règle OU unissant les deux modes précédents.

```
1.  let MSIZE = 2**32
2.  let REGNUM = 32
3.  type byte = int(8);
4.  type word = int(32);
5.  type index = card(5);
6.  type imm = card(6);
7.  mem M[MSIZE, byte]
8.  reg R[REGNUM, word]
9.  reg PC[1, word]
10. var TMP[1,byte]
11.
12. mode REGISTER(i : index) = R[i]
13.  syntax = format("r%d", i)
14.  image = format("01%5b", i)
15. mode IMMEDIATE(i : imm) = i
16.  syntax = format("#%d",i)
17.  image = format("1%6b",i)
18. mode ADDR = REGISTER | IMMEDIATE
19.
20. op ADD(dest:REGISTER, src1:REGISTER, src2:ADDR)
21.  syntax=format("add %s,%s,%s,dest.syntax,src1.syntax,src2.syntax)
22.  image=format("00000000000%s%s%s",dest.image,src1.image,src2.image)
23.  action= { dest = src1 + src2;}
24. op SUB(dest:REGISTER, src1:REGISTER, src2:ADDR)
25.  syntax=format("sub %s,%s,%s,dest.syntax,src1.syntax,src2.syntax)
26.  image=format("00000000001%s%s%s",dest.image,src1.image,src2.image)
27.  action= { dest = src1 - src2;}
28. op MOV(dest:REGISTER, src1:ADDR)
29.  syntax=format("mov %s,%s,dest.syntax,src1.syntax)
30.  image=format("11000000000%s%s0000000",dest.image,src1.image)
31.  action= { dest = src1;}
32. op ALU = ADD | SUB | MOV
33. op LDR(dest:REGISTER, src1:REGISTER)
34.  syntax=format("ldr %s,[%s],dest.syntax,src1.syntax)
35.  image=format("10000000%s%s0000000000",dest.image,src1.image)
36.  action= { dest = M[src1]}
37. op STR(src1:REGISTER, src2:REGISTER)
38.  syntax=format("str %s,[%s],src1.syntax,src2.syntax)
39.  image=format("10000001%s%s0000000000",src1.image,src2.image)
40.  action= { M[src2] = src1;}
41. op MEM = LDR | STR
42. op INST = ALU | MEM
```

Figure 1. Exemple de description nML (extrait)

2.1.3 Spécification des instructions

La description d'une instruction commence par le mot-clé `op` suivi d'un identifiant de l'instruction et d'une liste de paramètres typés (le type d'un paramètre peut être un type prédéfini, comme `card(4)` qui représente un nombre binaire codé sur 4 bits, ou un mode d'adressage défini dans la

description). Dans la Figure 1, l'instruction `ADD` a trois paramètres, tous spécifiés selon des modes d'adressage définis précédemment (ligne 20). Les deux attributs `syntax` et `image` indiquent la manière dont l'instruction est représentée, respectivement en langage d'assemblage et en binaire. Ces deux représentations font intervenir les représentations des opérandes telles que définies dans les modes d'adressage correspondants. La description d'une instruction comporte également un attribut `action` qui permet de spécifier la sémantique de l'instruction selon une syntaxe proche du langage C (ligne 23).

Des groupes d'instructions peuvent être définis par des règles OU (lignes 32, 41 et 42).

2.2 Génération d'un simulateur fonctionnel à partir d'une description nML

2.2.1 Vue générale

La Figure 2 présente le processus général : un jeu d'instructions est décrit par un utilisateur sous un formalisme donné (par exemple, le langage nML). Un outil génère automatiquement un simulateur fonctionnel pour ce jeu d'instructions, à partir de sa description. Le résultat peut être un outil indépendant ou bien se présenter sous la forme d'une bibliothèque qui peut être exploitée, par exemple, pour construire un simulateur de processeur.

Dans ce qui suit, nous allons expliquer, dans les grandes lignes, comment sont mises en place les deux grandes fonctions du simulateur fonctionnel : le décodage et l'émulation. Les approches décrites ici ont été mises en œuvre dans le cadre du développement d'une suite d'outils (parmi lesquels un générateur de simulateurs fonctionnels) à l'Institut de Technologie de Kanpur (Inde) [4]. Nous les avons reprises par la suite, lors de la conception de notre générateur GLISS.

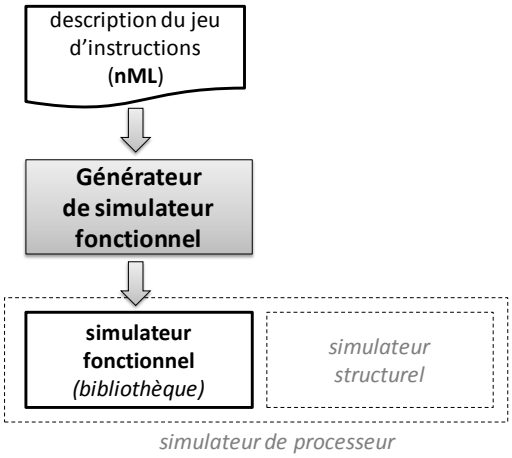

Figure 2. Génération automatique d'un simulateur fonctionnel

2.2.2 Construction d'un arbre de décodage

La fonction décodage du simulateur fonctionnel s'appuie sur un arbre de décodage. La première étape de la génération de cette fonction est donc la construction de cet arbre, que nous allons décrire dans ce qui suit. Nous utiliserons l'exemple de la Figure 1 pour illustrer le propos.

Dans un premier temps, le générateur parcourt la description pour en extraire la liste de tous les codes binaires possibles. Dans le cas d'une opération (mot-clé `op`) dont un des paramètres est défini par un mode d'adressage, tous les codages binaires possibles de ce paramètre sont pris en compte. Par

exemple, le troisième paramètre de l'opération ADD peut être un registre ou une valeur immédiate, ce qui correspond à deux codages différents. Le générateur calcule ces deux codes à partir de la description nML de l'opération et de celle des modes d'adressages. Le résultat de cette première étape est du type de ce qui est présenté dans le tableau ci-dessous : à chaque code binaire possible d'après la description nML, sont associés un masque qui indique les bits invariables et une valeur qui représente la valeur effective de ces bits.

id	mnémonique	masque	valeur
0	ADD_R	11111111111110000011000001100000	0000000000001xxxxx01xxxxx01xxxxx
1	ADD_I	11111111111110000011000001000000	0000000000001xxxxx01xxxxx1xxxxxx
2	SUB_R	11111111111110000011000001100000	0000000000101xxxxx01xxxxx01xxxxx
3	SUB_I	11111111111110000011000001000000	0000000000101xxxxx01xxxxx1xxxxxx
4	MOV_R	11111111111110000011000001111111	1100000000001xxxxx01xxxxx0000000
5	MOV_I	11111111111110000010000001111111	1100000000001xxxxx1xxxxxx0000000
6	LDR	11111111110000011000001111111111	1000000001xxxxx01xxxxx0000000000
7	STR	11111111110000011000001111111111	1000000101xxxxx01xxxxx0000000000

La construction de l'arbre de décodage se fait à partir des masques et des valeurs identifiés dans la première phase. Au nœud racine, on associe un masque de décodage égal à l'intersection (opération ET) des masques de tous les codes binaires. Dans notre exemple, le masque de décodage de la racine a ses dix bits de poids fort à 1 et les autres à 0. Il y a, en théorie, 1024 valeurs possibles pour ces dix bits mais, dans la liste des codes possibles, on ne retrouve que quatre de ces valeurs (en décimal : 0, 768, 513, 517). Un nœud fils est associé à chacune des valeurs effectivement rencontrées dans les codes d'instruction et on y attache la liste des codes concernés. Chaque nœud fils est ensuite traité, récursivement, comme le nœud racine (c'est-à-dire : construction du masque de décodage — en ignorant les bits du masque de décodage du nœud père —, recherche des valeurs possibles dans la liste des codes binaires associés au nœud, création des nœuds fils correspondants). Les nœuds associés à une seule instruction constituent les feuilles de l'arbre.

La Figure 3 présente l'arbre de décodage construit pour la description nML de la Figure 1. On notera que la construction de l'arbre de décodage permet de détecter que plusieurs instructions ont le même code binaire (même masque, même valeur) et donc de repérer certaines erreurs dans la description.

A chaque feuille de l'arbre de décodage sont associées une fonction de décodage, qui permet d'extraire les paramètres formels du code binaire de l'instruction, et une fonction de calcul qui permet de modifier l'état du processeur selon les modalités exprimées dans l'attribut `action` de la description nML de l'instruction.

2.2.3 Génération des fonctions de décodage et d'émulation du simulateur fonctionnel

La fonction décodage du simulateur fonctionnel exploite l'arbre de décodage. Au code instruction qui lui est passé en paramètre, elle applique le masque de premier niveau (nœud racine) et, selon la valeur obtenue, sélectionne un des nœuds fils (si la valeur ne correspond à aucune des valeurs de décodage des nœuds fils, la fonction renvoie un code d'erreur de type « instruction inconnue »). L'arbre est ainsi parcouru jusqu'à atteindre une feuille qui correspond à l'instruction recherchée. Les paramètres de l'instruction sont calculés à partir de la fonction de décodage associée au nœud.

La fonction émulation invoquée pour une instruction préalablement décodée lance l'exécution de la fonction de calcul associée à la feuille correspondante dans l'arbre de décodage. Cette fonction de calcul agit sur l'état du processeur émulé.

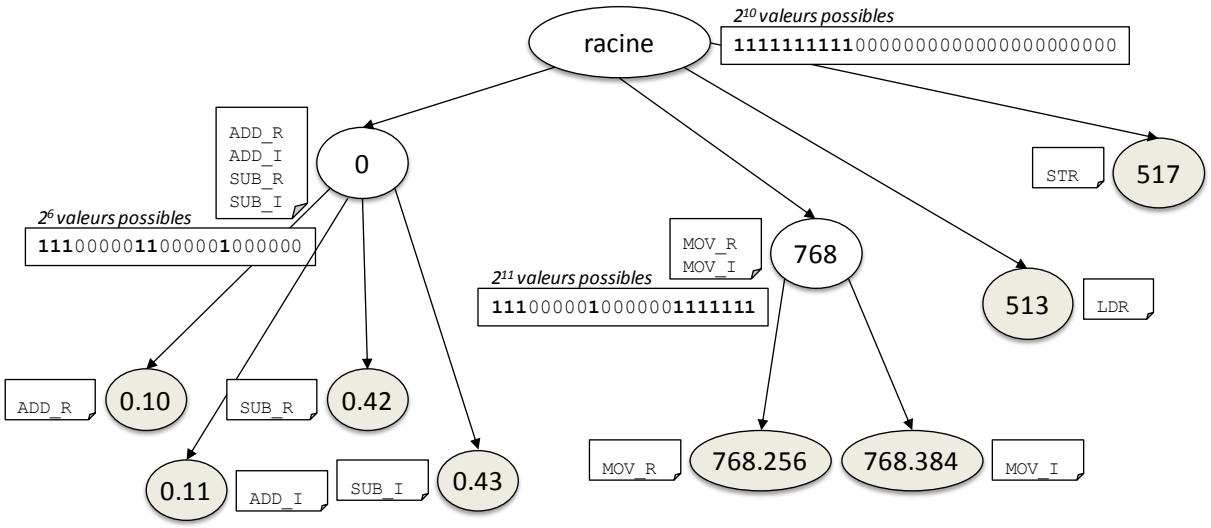

Figure 3. Arbre de décodage

2.3 Analyse des besoins

Ces dernières années, nous avons été amenés, dans le cadre de différents projets, à produire des descriptions de jeux d'instructions pour différentes architectures : ARM, Intel x86, PowerPC, HCS12, … Au travers de ces expériences, nous avons identifié un certain nombre de faiblesses du langage nML et des outils existants [4], que nous allons exposer ici.

2.3.1 Problèmes de désassemblage

Comme nous l'avons mentionné plus haut, la possibilité, pour un simulateur fonctionnel, de désassembler les codes binaires des instructions est loin d'être accessoire. Au contraire, elle constitue un élément clé pour implémenter des fonctions de débogage et de génération de traces dans un simulateur de processeur. Or, la description nML de certains jeux d'instructions ne permet pas de générer une fonction de désassemblage correct comme nous allons l'illustrer par quelques exemples.

Certains jeux d'instructions prévoient un encodage des opérandes immédiats. C'est le cas de l'ARM, dans lequel un opérande immédiat est exprimé sur 12 bits partagés en deux champs : un champ *rotation* (*r*) sur 4 bits et un champ *valeur* (*v*) sur 8 bits. La valeur de l'opérande est calculée en appliquant à la valeur *v* une rotation vers la droite de *2r* bits. Or, le langage nML ne permet pas de spécifier d'action de décodage pour l'attribut `syntax` : seule la valeur brute d'un paramètre peut être restituée. Aussi, le désassemblage d'une instruction ARM ayant un opérande immédiat est susceptible de faire apparaître une valeur erronée (ou en tout cas difficilement interprétable) de cet opérande (par exemple `#2817` ou `#0xB01`, au lieu de `#1024`).

Nous avons rencontré une autre difficulté en cherchant à produire un simulateur du jeu d'instructions HCS12X. En effet, ce jeu de données supporte des opérandes immédiats signés codés sur 9 bits. Le langage nML permet de définir un champ de 9 bits dans l'image binaire d'une instruction. Toutefois, les types définis en nML sont ensuite représentés, lors de la génération du

simulateur, par des types du langage C (8, 16 ou 32 bits). Un opérande 9 bits du jeu d'instructions HCS12X est donc extrait du code binaire d'une instruction comme spécifié par l'attribut `image` et stocké dans une variable C sur 16 bits. Or, dans la mesure où le bit de signe (bit n°8) n'est pas étendu, les valeurs négatives ne sont pas enregistrées correctement dans le simulateur. Elles peuvent être corrigées dans la fonction d'émulation (test du bit de signe) mais sont désassemblées de manière incorrecte (par exemple, la valeur immédiate `-3`, codée sur 9 bits par `111111101`, est désassemblée en `+509`).

Le jeu d'instructions x86 IA32 présente lui aussi des particularités qui compliquent le désassemblage. Par exemple, le registre accumulateur EAX peut n'être utilisé que partiellement : seulement ses 16 bits de poids faible (dans ce cas, il est désigné par AX), ou seulement un des deux octets de poids faibles (désignés par AH et AL). Le choix entre ces différentes possibilités dépend d'un préfixe situé en tête du code binaire de l'instruction (et dissocié du champ désignant le registre qui, dans les quatre cas, a pour valeur 000). La seule manière de décrire le jeu d'instructions pour que les instructions utilisant ce registre soient désassemblées correctement serait de décrire séparément toutes les versions d'une même instruction qui diffèrent par le préfixe. Ceci ne va évidemment pas dans le sens d'une simplification du processus de description.

Dans tous les exemples ci-dessus, les difficultés de désassemblage sont liées à l'absence, dans le langage nML, de la possibilité de spécifier des traitements (autres qu'une extraction pure et dure de champs de bits) pour le décodage des instructions. Nous allons proposer un peu plus loin une extension de ce langage permettant un décodage dynamique des paramètres.

2.3.2 Taille du code du simulateur fonctionnel généré

Comme nous l'avons expliqué précédemment, la construction de l'arbre de décodage implique une expansion systématique des codes binaires possibles pour chaque instruction. Or, pour chaque code binaire possible (feuille de l'arbre de décodage), il faut générer une fonction de décodage et une fonction d'exécution. La taille du code du simulateur fonctionnel est donc directement liée au nombre de feuilles dans l'arbre de décodage. A titre d'exemple, le code obtenu pour émuler le jeu d'instructions ARM 32 bits décrit en langage nML a une taille de 154 MO.

Or, dans cette multiplication des fonctions de décodage et d'émulation implique une redondance importante des calculs. En effet, lorsqu'un opérande d'une instruction peut être désigné selon plusieurs modes d'adressage, la fonction d'émulation de l'instruction est répliquée pour tous les codes binaires correspondants. Ainsi, ce qui est « factorisé » au moment de la description ne l'est plus au moment de la génération du code du simulateur.

Nous avons rencontré ce genre de problèmes en décrivant le jeu d'instructions ARM. La plupart des instructions ARM peuvent être exécutées de manière conditionnelle : les 4 bits de poids fort de leur code binaire indiquent la condition sous laquelle elles doivent être exécutées. En langage nML, il est possible de définir un type « condition » et d'intégrer le test de ce champ et la vérification de la condition associée dans la fonction d'émulation décrite dans l'attribut `action` de l'instruction. Toutefois, ceci ne permet pas un désassemblage convivial de l'instruction dans lequel l'identifiant de condition apparaîtrait en clair (par exemple, `addeq` ou `addne` ou `add` lorsque le code condition vaut respectivement `0000`, `0001` ou `1110`) plutôt que sous forme de valeur entière directement extraite du code binaire de l'instruction (soit `add0` ou `add1` ou `add14`). Pour un désassemblage correct, il faut décrire 16 modes « condition » différents, chacun ayant sa propre syntaxe (chaîne de caractères), comme illustré sur la Figure 4.

```
mode EQ()
   image = '0000'
   syntax = 'eq
mode NE()
   image = '0001'
   syntax = 'ne'
…
mode AL()
   image = '1110'
   syntax = ''
mode condition = EQ | NE | … | AL
op ADD(cond :condition)
   syntax=format("add%s …", cond.syntax)
   …
```

Figure 4. Description nML des instructions conditionnelles ARM.

Toutefois, cette description n'est pas satisfaisante en termes de taille du code généré. En effet, pour chaque instruction conditionnelle, ce sont 16 codes binaires différents qui vont être considérés dans l'arbre de décodage, entraînant une redondance inutile des fonctions de décodage et d'émulation.

Là encore, c'est le côté statique du langage nML qui ne permet pas de spécifier comment calculer la syntaxe d'un paramètre (par exemple le champ « condition ») à la volée.

On notera également que le traitement des conditions (identification et test) est à réaliser pour chaque instruction conditionnelle. Le langage nML donne la possibilité de définir des macros qui permettent de ne décrire ce traitement qu'une seule fois pour l'ensemble des instructions. Toutefois, à la génération de code, le contenu de la macro est répliqué pour chaque instruction. Pour réduire la taille du code généré, la possibilité de définir des fonctions serait un plus.

3. Vers un générateur de simulateur fonctionnel flexible et performant : le langage GLISS-nML et l'outil GLISS

3.1 Le langage GLISS-nML, une extension de nML

3.1.1 Variables locales et fonctions

Souvent, la description de la partie `action` d'une instruction nécessite l'introduction d'une variable temporaire intermédiaire. Dans le langage nML, les variables temporaires doivent être déclarées en tant que variables globales. Nous avons introduit la possibilité de déclarer des variables temporaires locales à la description d'une instruction. Ceci passe par la déclaration d'un pseudo-paramètre, spécifié comme un paramètre classique, mais avec un code binaire fixé à 0 (`0%b`), comme illustré sur la Figure 5 (variable `tmp`).

Nous avons également introduit la possibilité, pour l'utilisateur, d'accompagner la description GLISS-nML d'un jeu de fonctions (écrites en langage C) implémentant les calculs les plus fréquents (par exemple, le test d'une condition à partir du registre d'état). Ceci permet de réduire à la fois la taille de la description et la taille du code généré, puisque le simulateur intègre la fonction. Ceci est également montré sur l'exemple de la Figure 5 (fonction `foo`).

```
DESCRIPTION GLISS-nML
   op ABC(tmp:card(32), i :card(4), …)
      image = format("%0b%4b …", tmp.image, i.image, …)
      action = {
          tmp = foo(i);
      }
FONCTIONS EXTERNES
   int foo(int i) {
      …
   }
```

Figure 5. Pseudo-paramètres et fonctions utilisateur

3.1.2 Syntaxe dynamique des paramètres et prédécodage

L'élément clé de l'extension du langage nML en GLISS-nML réside dans l'introduction d'un nouvel attribut (nommé `predecode`) qui permet de spécifier comment les paramètres doivent être décodés dynamiquement. Il peut s'agir de déterminer la syntaxe d'un paramètre en fonction de sa valeur ou de celle d'autres paramètres, ou encore de calculer la valeur réelle d'un paramètre encodé.

Cette nouvelle fonctionnalité répond aux besoins identifiés dans la section précédente et permet à la fois d'obtenir un désassemblage correct des instructions et de maintenir, à la génération du code du simulateur, les factorisations réalisées lors de la description.

Les figures suivantes illustrent l'utilisation de cet attribut. La Figure 6 montre comment on peut spécifier, dans la description GLISS-nML, la manière dont la valeur d'un opérande immédiat doit être calculée pour une instruction ARM. La Figure 7 donne un exemple d'utilisation de l'attribut `predecode` pour générer une syntaxe correcte pour le champ condition d'une instruction ARM. Sur la Figure 8, on voit comment cet attribut permet de décoder dynamiquement la valeur d'un paramètre signé exprimé sur 9 bits, comme dans le jeu d'instructions HCS12X.

```
mode immediat (imm :card(32), rot:card(4), val:card(8))
   predecode {
      imm = coerce(32, val>>>2*rot) ;
   }
   syntax = format("#%d",imm)
   image = format("%4b%8b",rot,val)
op ADD(…, imm:immediat)
   syntax = format("add …,%s",…,imm.syntax)
   image = format("… %s",…,imm.image)
   …
```

Figure 6. Décodage dynamique de paramètres encodés (ARM)

```
mode condition (cond :card(4))
   predecode {
      switch (cond) {
          case 0 : cond.syntax = 'eq' ;
          case 1 : cond.syntax = 'ne' ;
          …
          case 14 : cond.syntax = '' ;
      }
   }
   syntax = format("%s",cond.syntax)
   image = format("%4b",cond)
op ADD(cond :condition)
   syntax=format("add%s …", cond.syntax)
   …
```

Figure 7. Description des instructions conditionnelles ARM en GLISS-nML

```
op DB(…, sign:card(1), val:card(8))
   predecode = {
      if sign == 0 then
          sign.syntax = '';
      else
          sign.syntax = '-';
          val = 512 - val;
      endif;
   }
   syntax=format("… %s %d",…,sign.syntax,val)
   image = …
   …
```

Figure 8. Décodage dynamique d'un nombre signé 9 bits (HCS12X)

3.2 GLISS : un générateur automatique de simulateurs fonctionnels

A partir de la description GLISS-nML d'un jeu d'instructions, GLISS génère automatiquement une bibliothèque de fonctions qui implémentent les fonctionnalités d'un simulateur fonctionnel. Cette bibliothèque peut être utilisée par un simulateur qu'il soit purement fonctionnel (auquel cas le simulateur est réduit à quelques dizaines de lignes) ou un simulateur structurel écrit en systemC par exemple. Les principales fonctions sont présentées dans le tableau ci-dessous.

`iss_fetch`	lit un code opération en mémoire
`iss_decode`	décode un code opération passé en paramètre et produit une représentation structurée de l'instruction
`iss_compute`	calcule le résultat de l'instruction passée en paramètre mais ne répercute pas ce résultat sur l'état du processeur (cette fonction est utile lorsque l'on construit un simulateur structurel pour un processeur qui exécute les instructions dans le désordre)
`iss_complete`	calcule le résultat de l'instruction passée en paramètre et met à jour l'état du processeur

D'autres fonctions permettent d'accéder en lecture ou en écriture à l'état logique du processeur (registres et mémoire) ou encore de désassembler une instruction (pour des besoins d'affichage).

GLISS intègre évidemment une fonction de chargement, dans la mémoire émulée, d'un code binaire au format `elf`.

3.3 Performance des simulateurs fonctionnels générés

Pour évaluer l'efficacité du langage GLISS-nML et de l'outil GLISS associé, nous avons identifié trois indices de performance : le temps de génération d'un simulateur à partir de la description du jeu d'instructions, la taille du code du simulateur généré et la vitesse de simulation. Dans cette section, nous présentons les résultats de mesures de ces indices pour la génération d'un simulateur du jeu d'instructions ARM v4. Nous avons considéré deux descriptions : l'une en langage nML et l'autre en GLISS-nML. Les résultats sont présentés dans la Table 1.

Les calculs ont été réalisés sur une machine bi-Opteron 240 (Linux Mandriva 32 bits 2.6.12-23 mdk #1), avec 2 GO de mémoire vive. Les performances du simulateur ont été mesurées lors de l'exécution des benchmarks `qsort` et `basicmath` (avec les jeux d'entrées `small` et `large`) extraits de la suite MiBenchs.

3.3.1 Temps de génération

Avec un générateur de simulateur dérivé des outils de Kanpur [4], et une description du jeu d'instructions ARM en langage nML, le temps de génération d'un simulateur fonctionnel est de 45 minutes. L'utilisation de notre outil GLISS (avec une description en GLISS-nML) permet de réduire ce temps à 3 minutes. Ce gain de temps est directement lié à la diminution du nombre de feuilles dans l'arbre de décodage (de 8 453 à 300). Cette réduction est liée à une factorisation du code grâce à l'utilisation de fonctions et au décodage dynamique. Comme indiqué plus haut, le seul fait de décoder dynamiquement le champ condition divise par 16 le nombre de feuilles générées pour chaque instruction conditionnelle.

3.3.2 Taille du code généré

La taille du code généré est elle-aussi en relation directe avec le nombre de feuilles dans l'arbre de décodage (codes binaires statiques). Grâce aux améliorations que nous avons apportées au langage, le code source passe de 154 MO à 3,4 MO. Le code binaire de la bibliothèque passe lui de 44MO à 720 kO. Dans les deux cas, le facteur est considérable.

3.3.3 Temps de simulation

Les mesures font également apparaître une réduction du temps de simulation de 4% à 27% sur les quatre tests considérés. Bien que la baisse soit modérée, ce résultat est notable car il n'était pas certain que la tendance soit celle-là. En effet, les modifications apportées au langage ont deux types d'effet sur le temps de simulation : d'un côté, le décodage statique est plus rapide puisque l'arbre de décodage est moins profond (réduction du nombre de feuilles) et donc parcouru plus facilement ; d'un autre côté, le décodage statique n'est pas complet, et certains paramètres doivent être traités dynamiquement. Il est satisfaisant de constater que le coût du décodage dynamique est plus que contrebalancé par le gain apporté à la partie statique. Notons également qu'une partie du gain peut être liée à la taille réduite du code, et donc à une meilleure localité dans le cache d'instructions.

		nML	GLISS-nML	*rapport*
nombre de feuilles de décodage		8 453	300	1/28
temps de génération		45 mn	3 mn	1/15
taille du code source généré		154 MO	3,4 MO	1/45
taille du binaire généré		44 MO	720 kO	1/61
temps de simulation	`qsort_small`	28,99s	21,17s	-26,9%
	`qsort_large`	536,11s	487,09s	-9,1%
	`basicmath_small`	527,32s	415,30s	-19,6%
	`basicmath_large`	5 850,56s	5606,56s	-4,0%

Table 1. Performances de l'outil GLISS (génération d'un simulateur fonctionnel ARM)

4. Conclusion

Dans cet article, nous avons décrit l'outil GLISS que nous avons développé afin de pouvoir générer automatiquement des simulateurs fonctionnels pour différents jeux d'instructions. La description d'un jeu d'instructions se fait en langage GLISS-nML, que nous avons dérivé du langage de description d'architectures nML. Nous avons utilisé l'outil GLISS pour générer des émulateurs pour des architectures très variées : RISC (ARM, TriCore, PowerPC), CISC (HCS12X, x86 en partie) et VLIW (Sharc). Les quelques mesures présentées dans l'article montrent que GLISS est bien plus efficace que des outils similaires existants [4] basés sur le langage nML.

Nous comptons poursuivre le développement de GLISS afin d'améliorer la vitesse de simulation et de pouvoir traiter un éventail encore plus large d'architectures. En particulier, nous sommes en train de mettre en place une solution pour gérer des jeux d'instructions multi-format (comme le jeu d'instructions ARM et son extension *Thumb* 16 bits).

Références

1. D. August, J. Chang, S. Girbal, D. Gracia Perez, G. Mouchard, D. Penry, O. Temam, N. Vachharajani – UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development – IEEE Computer Architecture Letters, 2007.
2. T. Austin, E. Larson, D. Ernst – SimpleScalar: an Infrastructure for Computer System Modeling – IEEE Computer, 35(2), 2002.
3. H. Cassé, P. Sainrat – OTAWA, a framework for experimenting WCET computations – 3rd European Congress on Embedded Real-Time Software, 2006.
4. Y. S. Chandra, R. Moona – Retargetable Functional Simulator Using High Level Processor Models – 13th International Conference on VLSI Design, 2000.
5. A. Fauth, J. Van Praet, M. Freericks – Describing instruction set processors using nML – European conference on Design and Test, 1995.
6. M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown – MiBench: a free, commercially representative embedded benchmark suite – IEEE International Workshop on Workload Characterization, 2001.